Memento Design Pattern

Last Updated : 18 Dec, 2024

The Memento Design Pattern is a behavioral pattern that helps save and restore an object's state without exposing its internal details. It is like a "snapshot" that allows you to roll back changes if something goes wrong. It is widely used in undo-redo functionality in applications like text editors or games.

Table of Content

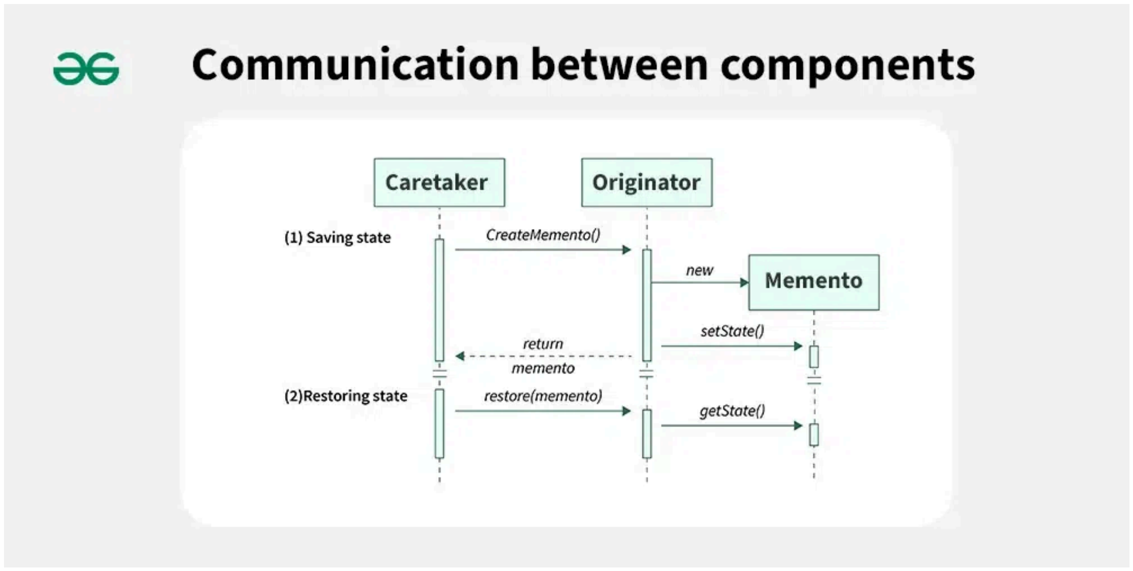
What is the Memento Design Pattern?

Without violating encapsulation, the internal state of an object can be captured to a prior one. You might wish to save checkpoints in your program as it develops so you can return to them at a later time.

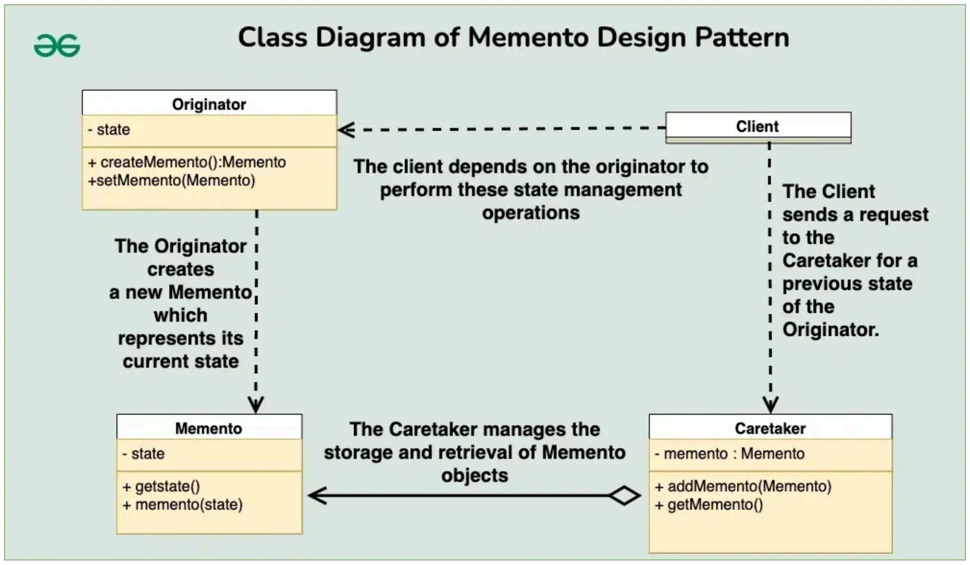
Components of Memento Design Pattern

- **Originator:**
    - The state of an object is established and maintained by this component.
    - It can create Memento objects to store its state and has methods to set and retrieve the object's state.
    - The Originator communicates directly with the Memento to create snapshots of its state and to restore its state from a snapshot.
- **Memento:**
    - The Memento is an object that stores the state of the Originator at a particular point in time.
    - It only provides a way to retrieve the state, without allowing direct modification. This ensures that the state remains
- **Caretaker:**
    - The Caretaker is responsible for keeping track of Memento objects.
    - It doesn't know the details of the state stored in the Memento but can request Mementos from the Originator to save or restore the object's state.
- **Client:**
    - Typically represented as the part of the application or system that interacts with the Originator and Caretaker to achieve specific functionality.
    - The client initiates requests to save or restore the state of the Originator through the Caretaker.

Communication between the components



1. **Client**: The client initiates the process by requesting the Originator to perform some operation that may modify its state or require the state to be saved. For example, the client might trigger an action like "save state" or "restore state."
2. **Originator**: The Originator either produces a Memento to save its current state (if the request is to save state) or retrieves a Memento to restore its prior state (if the request is to restore state).
3. **Caretaker**: The Caretaker acts as an intermediary between the client and the Originator, managing the Memento objects.


Class Diagram of Memento Design Pattern
Originator
- state
+ createMemento():Memento
+setMemento(Memento)
Client
The client depends on the originator to perform these state management operations
The Originator creates a new Memento which represents its current state
The Client sends a request to the Caretaker for a previous state of the Originator.
Memento
- state
+ getstate()
+ memento(state)
The Caretaker manages the storage and retrieval of Memento objects
Caretaker
- memento : Memento
+ addMemento(Memento)
+ getMemento()

- The caretaker calls the **createMemento()** method on the originator asking the originator to pass it a memento object.

- The caretaker maintains the memento object and performs the operation. In case of the need to undo the operation, the caretaker calls the **setMemento()** method on the originator passing the maintained memento object.
- The originator would accept the memento, using it to restore its previous state.

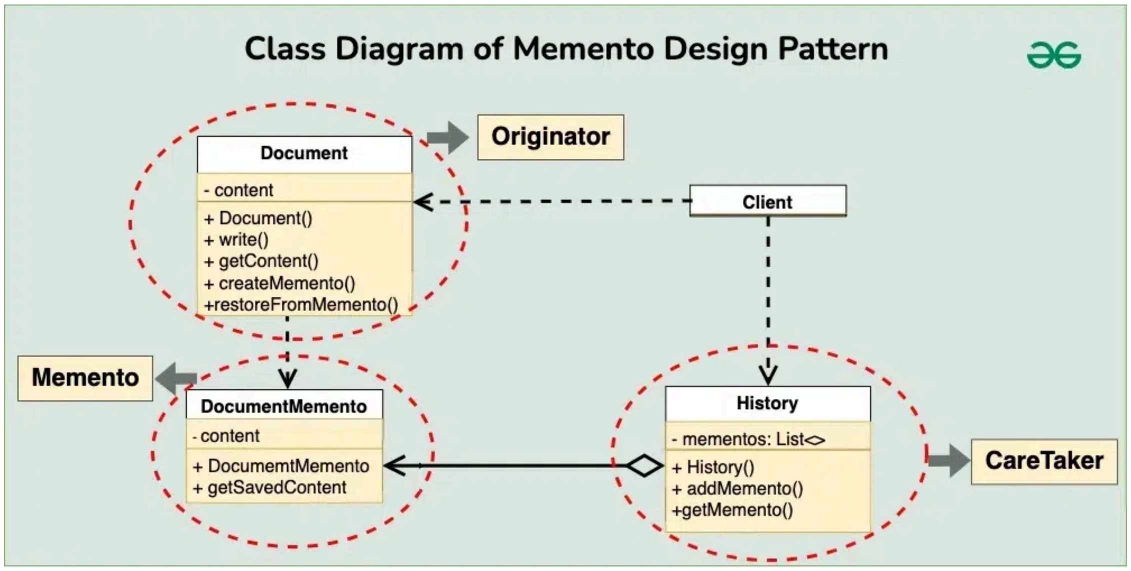
Memento Design Pattern Example(with implementation)

Problem Statement:

Consider creating a text editor application and want to include an undo feature that lets revert changes made to a document. The difficulty is in storing the document's state at different times and restoring it when required without disclosing the document's internal implementation.

Benefits of Using Memento Pattern in this scenario:

- **Encapsulation**: By using the Memento pattern, you can prohibit direct access to and manipulation of the document's state by encapsulating it within Memento objects.
- **Undo Functionality:** The Memento pattern makes it possible to construct an undo feature that lets users undo changes and recover earlier document states by saving snapshots of the document's state at various points in time.
- **Separation of Concerns:** By separating state management responsibilities from the document itself, the Memento design encourages code that is clearer and easier to maintain.



Below is the implementation of the above problem

Let's break down into the component wise code:

1. Originator (Document)

```
public class Document {
    private String content;

    public Document(String content) {
        this.content = content;
    }

    public void write(String text) {
        this.content += text;
    }

    public String getContent() {
        return this.content;
    }

    public DocumentMemento createMemento() {
        return new DocumentMemento(this.content);
    }

```

```java
        this.content = memento.getSavedContent();
    }
}
```

2. Memento

```java
public class DocumentMemento {
    private String content;

    public DocumentMemento(String content) {
        this.content = content;
    }

    public String getSavedContent() {
        return this.content;
    }
}
```

3. Caretaker (History)

```java
import java.util.ArrayList;
import java.util.List;

public class History {
    private List<DocumentMemento> mementos;

    public History() {
        this.mementos = new ArrayList<>();
    }

    public void addMemento(DocumentMemento memento) {
        this.mementos.add(memento);
    }

}
```

Complete code for the above example

Below is the complete code for the above example:

```
import java.util.ArrayList;
import java.util.List;

// Originator
class Document {
    private String content;

    public Document(String content) {
        this.content = content;
    }

    public void write(String text) {
        this.content += text;
    }

    public String getContent() {
        return this.content;
    }

    public DocumentMemento createMemento() {
        return new DocumentMemento(this.content);
    }

    public void restoreFromMemento(DocumentMemento memento) {
        this.content = memento.getSavedContent();
    }
}

// Memento
class DocumentMemento {
    private String content;

    }

    public String getSavedContent() {
        return this.content;
    }
}

// Caretaker
class History {
    private List<DocumentMemento> mementos;

    public History() {
        this.mementos = new ArrayList<>();
    }

    public void addMemento(DocumentMemento memento) {
        this.mementos.add(memento);
    }

    public DocumentMemento getMemento(int index) {
        return this.mementos.get(index);
    }
}

public class Main {
    public static void main(String[] args) {
        Document document = new Document("Initial
content\n");
        History history = new History();

        // Write some content
        document.write("Additional content\n");
        history.addMemento(document.createMemento());

        // Write more content
        document.write("More content\n");
        history.addMemento(document.createMemento());

        // Restore to previous state
        document.restoreFromMemento(history.getMemento(1));

        }
    }
```

Output

```
Initial content
Additional content
More content
```

When to use Memento Design Pattern?

- **Undo functionality**: When your application needs to include an undo function that lets users restore the state of an object after making modifications.
- **Snapshotting**: When you need to enable features like versioning or checkpoints by saving an object’s state at different times.
- **Transaction rollback**: When there are failures or exceptions, like in database transactions, and you need to reverse changes made to an object’s state.
- **Caching**: When you wish to reduce duplicate calculations or enhance efficiency by caching an object’s state.

When not to use Memento Design Pattern?

- **Large object state**: Keeping and managing several snapshots of an object’s state might use a lot of memory and computing power if its state is big or complicated.
- **Frequent state changes**: It may become impractical or wasteful to save and manage snapshots of an object’s state if its state changes regularly and unexpectedly.
- **Immutable objects**: The Memento pattern may not be very useful for capturing and restoring an object’s state if it is easily reconstructable or immutable.
- **Overhead**: The codebase may become more complex if the Memento pattern is included, particularly if the application does not need capabilities like state rollback or undo functionality.